Promised Consistency for Rollback Recovery

Denis Caromel, Christian Delbé, Ludovic Henrio

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Promised Consistency for Rollback Recovery

Denis Caromel — Christian Delbé — Ludovic Henrio

N° 5902

May 2006

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5902--FR+ENG

Promised Consistency for Rollback Recovery

Denis Caromel , Christian Delbé , Ludovic Henrio

Thème COM — Systèmes communicants
Projet Oasis

Rapport de recherche n° 5902 — May 2006 — 29 pages

Abstract: Checkpointing protocols usually rely on the constitution of consistent global states, from which the application can restart upon a failure. This paper proposes a new characterization and technique to build a recoverable state, aiming at relaxing the constraints and overhead.

$\mathcal{P}$-consistency, for *Promised consistency*, is proposed as such a recovery condition on a global state. A key idea is to use *promised events*: place holders forcing any restart to reach an actual global state of the first execution. A preliminary contribution is a formal treatment of *potential causality*, studying its impact on recoverability and determinism.

Key-words: Rollback recovery, event-based, potential causality, formal model, application, consistency

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 —Télécopie : +33 4 92 38 77 65

Cohérence promise pour le recouvrement arrière

Résumé : Les protocoles de tolérance aux pannes par point de reprise constituent durant l'exécution des états globaux cohérents, depuis lesquels l'application peut repartir après une panne. Ce rapport propose une nouvelle technique pour constituer des états globaux non cohérents mais recouvrables, permettant de relacher les contraintes de création d'états.

La $\mathcal{P}$-consistency, pour *cohérence promise*, est prosposée comme condition de recouvrabilité d'un état global. Cette condition repose sur l'utilisation d'*évenements promis*: des conteneurs d'événements qui forcent la réexecution jusqu'à atteindre un état global cohérent de la première exécution. Comme contribution préliminaire, nous proposons une étude formelle de la causalité potentielle, ainsi qu'une analyse de son impact sur les notions de recouvrement arrière et de déterminisme.

Mots-clés : Recouvrement arrière, modélisation événementielle, causalité potentielle, modèle formel, application, cohérence

1 Introduction

Rollback-recovery fault-tolerance relies on regularly logging enough data about the processes of an application in order to recover one or several processes after a failure [9]. A fault-tolerance protocol must ensure that such data constitute a *recoverable* state: a state from which all or a part of the application can restart if a failure occurs. Checkpointing protocols usually build consistent global states because these states are recoverable. Those protocols suppose that a checkpoint can be triggered at any time on any process of the application. However, forcing checkpoints at a given time could cause significant performance degradation, due to contention on the stable storage [22] or to variation of the size of the data that must be checkpointed [11]. In the worst case, the processes may not be able to checkpoint at any time [2]. For all those reasons, in various cases, a triggered checkpoint *must* or *should* be postponed until it is possible. We aim at contributing to such situations with the ability to recover from a non-consistent state, without global synchronization, without systematically logging event application data.

Within the remaining of this general introduction, the paper approaches the related work in a structured manner, first discussing causality in distributed systems, continuing with recoverability, and causal relation between events. Subsequently, objective and roadmap of the paper are discussed. The bulk of the paper is structured in four main sections. Section 2 introduces key definitions and basic properties, including potential causality and deterministic events. Section 3 presents the promise-based recovery. The paper proves that an execution restarted from a P-consistent state will always reach an actual global state of the first execution. Finally, Section 4 exhibits an application, the $\mathcal{P}$-consistency of the created global states for a protocol ensure the correctness of an existing fault-tolerance protocol for active objects.

1.1 Causality in Distributed Systems

Lamport introduced in [14] the concept of one event happening before another in a distributed system, i.e., *virtual or logical time*. He defined the well-known *happened-before* relation that partially orders events of a distributed execution, and subsequently proposed the first snapshot algorithm [4].

Based on this work, Mattern focused the "causal domain", in opposition to the "time domain", focusing on dependencies between events. He stated that events are causally related, and proposed in [17] a formal model for characterizing a distributed execution as a set of events ordered by a *causality relation*, and chose the Lamport's happened-before relation as the causality relation. He also extended the idea of global snapshot in [4] and defined the *cuts* and *consistent cuts*.

Those seminal about causal dependency are an important foundation for specifying and proving distributed programs; proving the recoverability of a system after a failure is usually based on the causality relation between processes. In particular, the idea of consistency is one of the key principles of rollback-recovery fault-tolerance in distributed systems.

1.2 Recoverability in Distributed Systems

We can distinguish two main approaches for rollback-recovery fault tolerance. The first, consistent checkpointing [12], consists in building global states during the execution. Global states comprise one checkpoint per process, and a protocol ensures that some global states are *consistent*. Intuitively, a global state is consistent if it may occur during a failure free execution. Such a global state is obviously recoverable as it is a possible state of the system.

The first protocol proposed to create a global state is defined in [4]; it has induced a family of protocols named *coordinated* or *synchronous* checkpointing protocols, characterized by an explicit synchronization phase between processes for coordinating local checkpoints. On the other hand, the communication induced checkpointing protocols [16] do not rely on any explicit synchronization phase; the messages of the application are used to coordinate local checkpoints.

The second approach for ensuring the recoverability of a state is *message-logging*. The built states do not need to be consistent anymore: the recoverability is ensured by logging all non-deterministic events occurring on each process. This technique relies on the *piecewise-determinism* assumption: a process execution can be modeled as a sequence of deterministic state intervals separated by *observable* non-deterministic events [18]. In [13], the authors state that a state is recoverable if all component process states are logged and the resulting system state is consistent. In other words, a recoverable state is constituted of one or several checkpoints with enough informations about non-deterministic events.

There are three kinds of message-logging protocols. The pessimistic message logging protocols synchronously log on a stable storage all the non-deterministic events. Each pair (checkpoint+logged events) is then a recoverable state.

The optimistic message-logging, does not force the synchronous logging of non-deterministic events. Those logs can be postponed, which can lead to orphan processes, i.e. processes causally dependent of not-yet-logged events. A recoverable state is then constituted of a pair (checkpoint+logged events) and of the pair (checkpoint+logged events) of all the orphan processes, recursively. In both pessimistic and optimistic message-logging, the non-deterministic events must be finally logged on a stable storage.

The causal message-logging combines the advantages of both pessimistic and optimistic approaches. It avoids orphans processes [1], without enforcing synchronous logging of non-deterministic events. This is realized by piggybacking information on the non-deterministic events on messages of the application, and storing them in volatile memory until they are logged on a stable storage. Application data are supposed to be available; in practice, such data must be logged, at least in volatile memory. The needed information for recovery is then scattered in all the system: a recoverable state is constituted of a single checkpoint, and a set of non-deterministic events that must be gathered from all the system.

1.3 Causality Relation Between Events

Originally, the Lamport relation was intended to model the *logical time* in a distributed execution. Mattern observed that this relation also models an approximation of the causality relation between events: if $e \prec^{hb} e'$, then e can *potentially* affect the event e'. However, this relation does not systematically characterize the *true* causality between events. We identify below how the happened-before relation can be extended to get closer to this true causality.

The causality induced by the happened-before relation supposes that two consecutive events on the same process are inevitably causally linked, since events that occur on the same process are *totally ordered*. This assumption leads to the concept of *false causality* introduced by Tarafdar in [20]: an event that happens before another one does not necessarily cause it. The authors then distinguish Lamport's *logical time* from *potential* and *true* causality: "logical time is not potential causality". In other words, the Lamport's happened-before relation is sufficient, but *not necessary*, to characterize causality relation between events in every physically possible systems. Even if the true causality relation in a distributed execution is not observable, a more accurate potential causality between events can be tracked during the execution, as in [21].

The application or the system may ensure a given order of message delivery, the causal order must. A potential causal order must also take into account such a *message ordering*. The message ordering can be expressed as additional causality relations between message delivery events [5]. Not considering message delivery ordering can lead to impossible, thus unrecoverable, states.

Message significance also relates to a more accurate potential causality relation [15, 19]. Indeed, a message reception is not systematically causally related to all the other events on the receiver. Those papers exploit possible message reordering to enhance exisiting fault-tolerance protocols.

To summarize, the happened-before relation does not always characterize accurately the causality relations between the events of an execution. Considering more accurate causality relations, and then reducing the causality dependencies that characterize an execution is an important concern for developing fault-tolerance mechanisms [6]. Indeed, false causality relations can lead to restrictive and superfluous requirements on the recoverability condition.

1.4 Objectives and Road-map

The objective of this paper is to present the formal principles allowing to recover from a non-consistent global state without having logged any event and without relying on a global synchronization. We present our solution, namely the $\mathcal{P}$-consistency, a new recoverability condition on a global state. The correctness of the recovery relies on the ability to force the re-execution from this state and finally reach a consistent global state of the first execution. In general, such a control on an execution, necessitates to remember the events that occurred between the inconsistent state and the consistent state. Under the piecewise-determinism assumption, it is sufficient to remember only the non-deterministic events and the exact moment when they occurred, that is to say their position relatively to all the other events.

This mechanism could be very space-consuming if the non-deterministic events are numerous and the data are voluminous, as data produced by the application have to be remembered. Message-logging mechanisms rely on this solution; their performances dramatically decrease with the number and the size of exchanged messages.

To avoid logging application data, Section 3.1 introduces the concept of a *promised event*, a place holder for a non-deterministic event. When the corresponding event occurs during the re-execution, it is forced by the promised event to occur identically and at the same moment as in the first execution: the promised event is replaced by the original one. *Wait-by-necessity* is a synchronization mechanism that applies to promised event; it ensures the consistency of a local execution. This synchronization ensures that all the causal past of a deterministic event must have been executed *before* executing this event. In practice, if a process tries to execute a deterministic event that causally depends on a promised one, then the *local* execution is blocked in a wait-by-necessity state until the promised event is replaced.

Based on promised events, Section 3.2 defines $\mathcal{P}$-consistent (*promised* consistent) states which are intermediate states of the application during the recovery. Section 3.4 shows that all possible executions starting from a $\mathcal{P}$-consistent state finally reach a global state of the first execution, without having to systematically remember application data; such data are ensured to be produced by the re-execution. The wait-by-necessity synchronization also guarantees in a lazy and local manner the correct ordering of events during re-execution. Thus, the application can be recovered provided an initial $\mathcal{P}$-consistent global state can be generated from an inconsistent state with enough positioned promised events to force the execution up to a consistent state. The recovery only consists in placing the deterministic events.

The $\mathcal{P}$-consistency is defined over a general potential causality relation characterized in Section 2.2 which reduces the relations between events. Indeed, with a Lamport causal order, promised events should be exactly ordered relatively to all the other events; partial order introduced by potential causality allows promised events to be just ordered between themselves.

To summarize, this paper formally defines the impacts of an accurate potential causality on the notion of recoverability and determinism. Thanks to this formalism, we demonstrate in Section 3 the equivalence between a recovery from a $\mathcal{P}$-consistent global state and a recovery from a consistent global. Finally, we consider a real application of the $\mathcal{P}$-consistency: we developed a fault-tolerance protocol for the ASP calculus and its Java implementation – ProActive. The constrained checkpointability of ProActive does not allow the creation of consistent global states; we implemented a protocol that creates $\mathcal{P}$-consistent global states and allows to recover from those states. Section 4 defines the potential causality induced by the semantic of the ASP model, and formally proves the $\mathcal{P}$-consistency of the created states.

The main contributions of this paper are:

- a formal study of the impact of an accurate causality on recoverability and determinism,

- the definition of promised events,
- the definition of a new recovery condition, namely the $\mathcal{P}$-consistency,
- a formal proof of the recoverability from a $\mathcal{P}$-consistent global state,
- and finally a real usecase of the $\mathcal{P}$-consistency: a fault-tolerance protocol for ASP.

2 Formalism

We consider distributed systems consisting in several processes $P_1, ..., P$ that communicate only by message passing with any message ordering. An execution can be viewed by an external observer as a sequence of global states S, with $S = \{s_1, ..., s\}$ a set of local state s_i (one for each process P_i), following a given reduction. Each step of the reduction is characterized by an event e as follows: $S \xrightarrow{e} S'$. This reduction $\xrightarrow{e}$ represents the *execution* of the event e. Let us denote as follows a given execution from a global state S^0 to another global state S^n; when e^i are not necessary, we use the notation $S^0 \xrightarrow{*} S^n$

$$S^0 \xrightarrow{e^1 ... e^n} S^n \Leftrightarrow S^0 \xrightarrow{e^1} S^1 \dots S^{n-1} \xrightarrow{e^n} S^n$$

We suppose that an event is associated with a single process, denoted $e \in P_i$: each event modifies a single local state, and each local state is uniquely characterized by the ordered sequence of events that occurred on its process (where $\exists!$ means "there is a unique"):

$$S \xrightarrow{e} S' \Rightarrow \exists! k, S = \{s_1, ..s_k .., s_n\} \wedge S' = \{s_1, ..s'_k .., s_n\}$$

Considering the set of all possible executions from a given initial state, we define an equivalence relation between those possible executions. This definition depends on a notion of equivalence on states ($\equiv$); in a first time, this equivalence can be the strict equality between states.

Definition 1 (Execution equivalence) *Two executions* $S \xrightarrow{e^1} S_1^1 \dots S_1^{n-1} \xrightarrow{e^n} S_1^n$ *and* $S \xrightarrow{e^{\sigma(1)}} S_2^1 \dots S_2 \xrightarrow{e^{(\sigma(n)}} S_2^n$ *are said to be* equivalent *if,* σ *is a* permutation*, and every two global states* S_1^i *and* S_2^i *obtained after executing the same subset of events* $\{e^1, \dots, e^n\}$ *in a different order are equivalent:*

$$\forall i \leq n, \left(S \xrightarrow{e^1 ... e^i} S_1^i \wedge S \xrightarrow{e^{\sigma(1)} ... e^{\sigma(i)}} S_2^i \wedge \forall j \leq i, \sigma(j) \leq i \right) \Rightarrow S_1^i \equiv S_2^i$$

2.1 Happened-before relation

Lamport introduces in [14] the concept of one event happening before another in a distributed execution, and defines a partial ordering between events. To identify communication events, let the set $\Gamma = \{(e, e') \in \{e^1 \dots e^n\} * \{e^1 \dots e^n\}\}$ be such that e is the sending of a given message and e' the reception of this message for a given execution $S \xrightarrow{e^1 ... e^n} S'$.

Definition 2 (Happened-before relation) *Let local events be totally ordered by the execution order on P_i: $\prec_i^{hb}$. $\prec^{hb}$ is a partial ordering such that $e \prec^{hb} e'$ if and only if*

$$e \prec_i^{hb} e' \quad \vee \quad (e, e') \in \Gamma \quad \vee \quad (\exists e'', e \prec^{hb} e'' \prec^{hb} e')$$

This relation models the *observable* precedence order between events, or *logical time*: local events are totally ordered regarding the execution order $\prec_i^{hb}$. Moreover, events that occurred before the sending of a message also occurred before the reception of this message. Then, Mattern [17] presents a formal model for characterizing a distributed execution by an *event structure*. An event structure consists in a set of events partially ordered by the Lamport's happened-before relation:

Definition 3 (Execution characterisation) *Let $S \xrightarrow{e^1 \dots e^n} S'$ be an execution, $E = \{e^1 \dots e^n\}$ the set of executed events and $\prec^{hb}$ the happened-before relation. The event structure $(E, \prec^{hb})$ characterizes $S \xrightarrow{e^1 \dots e^n} S'$.*

The concept of equivalent executions (Definition 1) is related to this characterization of an execution by a partially ordered set (or poset for short). Indeed, the *observed* execution $S \xrightarrow{e^1 \dots e^n} S'$ characterized by $(E, \prec^{hb})$ is *one* of the several possible linear extensions of $(E, \prec^{hb})$; and all the executions that are linear extensions of $(E, \prec^{hb})$ are equivalent.

2.2 Potential Causality

As shown in Section 1.3, the Lamport relation leads to false causality between events. The happened-before relation is in fact the *most general* causality relation; it models any existing distributed system communicating by message passing. In particular, the causal relation between the sending and the reception of a message is obvious for all physically realistic systems; a message is sent before being received. It also captures any causal relation between receptions on a process due to a specific message ordering, since those receptions are totally ordered.

Even if the true causality between events cannot be formally characterized [20], it can be approximated; such approximations are called *potential causality*. A potential causality relation can take into account the specifities of a system in order to represent causality relation between events. It must also capture the influence of a specific message ordering on an execution; the most general solution would be to totally order all the receptions. We define here the requirements for a potential causality relation:

Definition 4 (Potential causality) *A partial order $\prec$ is a potential causality order if it is sufficient to characterize equivalent executions: two executions only permuting non-potentially causally ordered events must be equivalent:*

$$\textit{if } \prec \textit{ is a potential causality order then} \forall S, S_1, S_2, \sigma,$$

$$\begin{cases} S \xrightarrow{e^1 \ldots e^n} S_1 \wedge S \xrightarrow{e^{\sigma(1)} \ldots e^{\sigma(n)}} S_2 \\ \sigma \text{ is a permutation preserving } \prec \end{cases} \Rightarrow S_1 \equiv S_2$$

Two *executions* are said to be *potentially equivalent* if they consist of the same event, ordered in the same way according to the chosen potential causality. Potential equivalence of two executions implies their equivalence, because potential causality is *sufficient* for characterizing executions.

2.3 From Compatibility to Potential Causality

The false causality issue is due to the total local order $\prec_i^{hb}$. Indeed, a total order represents a given local execution on a process as can be traced. On the other hand, in most cases, the exact order in which instructions are executed is not observable, and if two events do not conflict, i.e. if executing one after or before the other has no consequence, then there is no need to strictly order them. Those two events can be safely exchanged; the two resulting executions are potentially equivalent. For example, two consecutive reads of the same variable can be safely exchanged, without altering the behavior of the execution. If the semantics of the system states that two events can be exchanged, they are called *compatible*:

Definition 5 *Two events e and e' are compatible, if both can be executed before the other and the effect of their execution is independent of the order in which they are executed. More formally:*

$$e \bowtie e' \Leftrightarrow \begin{cases} \left(\exists S, S_1, S_2,\ S \xrightarrow{ee'} S_1 \wedge S \xrightarrow{e'e} S_2\right) \ \wedge \\ \left(\forall S', S_1, S_2,\ S' \xrightarrow{ee'} S_1 \wedge S' \xrightarrow{e'e} S_2 \Rightarrow S_1 \equiv S_2\right) \end{cases}$$

Based on the compatibility relation between events, a more accurate potential causality relation $\prec$ can be inferred. The compatibility relation takes into account the specificities of the system and its semantics. The definition below shows how a potential causality relation $\prec$ can be inferred from the knowledge of compatible events, and with the restriction that, as in the happened-before relation, a message is received after being sent.

Definition 6 (From compatibility to potential causality) *For a given execution $(E, \prec^{hb})$, the order defined by the transitive closure of $\prec^{hb}$ and all the permutations allowed by the compatibility relation $\bowtie$:*

$$e \prec e' \text{ iff } \left(e \prec^{hb} e' \wedge \neg(e \bowtie e')\right) \vee (\exists e'', e \prec e'' \wedge e'' \prec e')$$

Equivalently, provided that the sending of a message is not compatible with its reception:

$$e \prec e' \text{ iff } \left(S \xrightarrow{*} S^1 \xrightarrow{e} S^2 \xrightarrow{*} S^3 \xrightarrow{e'} S^n \wedge \neg(e \bowtie e')\right) \vee (e \prec e'' \wedge e'' \prec e')$$

$\prec$ defined above is a potential causality order.

2.4 Determinism

Let the operator $\backslash$ be defined as follows: $E \backslash E' = \{e | e \in E \wedge e \notin E'\}$, and "$e$ is minimal in A" mean $\forall e' \in A, e' \prec e \Rightarrow e' = e$

Considering a potential causal order and the compatibility relations between events allows us to define a predicate that identify deterministic events. An event is said to be deterministic if, once its execution is possible, it necessarily happens and have the same effect on the system in every possible execution.

Definition 7 (Deterministic Events) *Let $Det(e)$ be a predicate such that for all states S of an execution $(E, \prec)$, such that $S_0 \xrightarrow{e_1..e_n} S$, for any reduction $\xrightarrow{e}$ executed from the state S, if $Det(e)$ then e necessarily happen after S in one of the executions represented by $(E, \prec)$.*

$$\forall S', S \xrightarrow{e} S' \Rightarrow \neg Det(e) \vee (e \in E \backslash \{e_1..e_n\} \wedge e \text{ is minimal (for } \prec) \text{ in } E \backslash \{e_1..e_n\})$$

Equivalently, because of the interplay between potential causality and equivalent executions:

$$Det(e) \Rightarrow \forall S, e', S', S'' \left(S \xrightarrow{e} S' \wedge S \xrightarrow{e'} S'' \Rightarrow \exists S_1, S_2, S' \xrightarrow{e'} S_1 \wedge S'' \xrightarrow{e} S_2 \wedge S_1 \equiv S_2 \right)$$

This formalization of deterministic events is particularly adapted to potential causality. Indeed, an event is deterministic if it is *compatible with every event that could be performed instead of it.*

Property 2.1 (Determinism and Compatibility) $S \xrightarrow{e} S' \wedge Det(e) \Rightarrow \forall e' (S \xrightarrow{e'} S'' \Rightarrow e \bowtie e')$

Note that it is always safe to consider a deterministic event as non-deterministic.

2.5 Cuts

As defined by Mattern in [17], a cut of an execution is a partially ordered set defined as follows:

Definition 8 (Cut) *A cut $\mathcal{C}$ of an execution $(E, \prec)$ is a finite subset $\mathcal{C} \subseteq E$ such that*

$$\forall e, e' \in E, e \in \mathcal{C} \wedge e' \prec_i e \Rightarrow e' \in \mathcal{C}$$

A global state S^i is actually a particular cut, or *consistent* cut of an execution.

Definition 9 (Consistent cut) *A cut $\mathcal{C}$ is consistent if and only if $e \in \mathcal{C} \wedge e' \prec e \Rightarrow e' \in \mathcal{C}$*

Considering $\prec$ as a potential causality order, a consistent cut $\mathcal{C}$ correspond to *several* equivalent states: if two states correspond to the same consistent cut, then they are equivalent. The set of consistent cuts of an execution represents the set of possible global states of all the potentially equivalent executions. Let $\mathcal{C}$ be a consistent cut of an execution $(E, \prec)$ and S be the state corresponding to $\mathcal{C}$. If $S \xrightarrow{e} S'$ is a reduction of $(E, \prec)$ then $\mathcal{C}' = \mathcal{C} \cup \{e\}$ is also a consistent cut of $(E, \prec)$. Decreasing the strictness of $\prec$ increases the set of consistent cuts. The set of *cuts of potentially equivalent executions* form a lattice.

3 Characterizing and Enforcing Executions

We present in this section how, thanks to promised events, an execution can be controlled. We then define and demonstrate the recoverability of a $\mathcal{P}$-consistent cut. Main proofs related to this section, and some additional properties related to Section 3.5 can be found in appendix.

3.1 Promised Events

A promised event is a place-holder for a non-deterministic event that has been performed during the first execution. A promised event can be causally ordered relatively to other events, and holds enough information to turn a non-determinate event into a determinate one during the re-execution: when an event corresponding to a promise should occur, it is transformed into the same event that occurred in the first execution. In practice, an event can be turned into a promised one if it can be identified and isolated.

The content of e may be stored or not in the corresponding promised $\mathcal{P}(e)$ depending on whether this content is ensured to be regenerated or not. If sending a message is determinate, we show that messages are inevitably resent during the re-execution: the data communicated by a message should not be stored. On the other side, if e consists in choosing a random number, $\mathcal{P}(e)$ must then store the obtained value since it will not regenerated during the re-execution.

A promised event is subject to *wait-by-necessity* mechanism, ensuring that the local execution is blocked if a deterministic event that causally depends on a promised should be executed. Once the event corresponding to this promised one occurs, the execution continues.

We define the operator $\lhd$, being used as follows: $\mathcal{P}(e) \lhd e'$ iff e' is a non-deterministic event corresponding to e (it should be unique for each execution), in that case e' should be forced to occur identically to e and at its position. Globally, a promised event $\mathcal{P}(e)$ ensures that $\xrightarrow{\mathcal{P}(e)e_2..e_n e'}$ is equivalent to $\xrightarrow{ee_2..e_n}$, provided $\mathcal{P}(e) \lhd e'$ and $\mathcal{P}(e) \not\lhd e_i$ for all e_i. We require that $\mathcal{P}(e) \lhd e$.

Promised events mechanism relies on the assumption that the execution is piecewise-deterministic, that is to say each non-deterministic choice is represented by an identifiable event.

Property 3.1 (Piecewise-determinism) *Non-deterministic events are always identifiable:*

$$S \xrightarrow{e} S' \wedge \neg Det(e) \wedge S \xrightarrow{e'} S'' \Rightarrow S'' \xrightarrow{e} S''' \vee e' \lhd \mathcal{P}(e)$$

Generally, e denotes a non-promised event, $\mathcal{P}(e)$ is a promised one and $e_{\mathcal{P}}$ can be either $\mathcal{P}(e)$ or e. Compatibility is trivially extended to promised events as follows:

$$e \bowtie e' \Rightarrow \mathcal{P}(e) \bowtie e' \ \wedge \ e \bowtie \mathcal{P}(e') \ \wedge \ \mathcal{P}(e) \bowtie \mathcal{P}(e')$$

3.2 $\mathcal{P}$-cuts

A $\mathcal{P}$-cut is a cut that can contain some promised events. We first introduce the following notation:

$$e \in_{\mathcal{P}} \mathcal{C}_p \Leftrightarrow e \in \mathcal{C}_p \vee \mathcal{P}(e) \in \mathcal{C}_p$$

Definition 10 ($\mathcal{P}$-cut) *$\mathcal{C}_p$ is a $\mathcal{P}$-cut of an execution $(E, \prec)$ if and only if $e \in_{\mathcal{P}} \mathcal{C}_p \Rightarrow e \in E$ and:*

$$\forall e, e',\ (e \in_{\mathcal{P}} \mathcal{C}_p), e' \prec_i e \Rightarrow \big(Det(e) \wedge e' \in_{\mathcal{P}} \mathcal{C}_p) \vee (\neg Det(e) \wedge (e' \in_{\mathcal{P}} \mathcal{C}_p \vee Det(e'))\big)$$

Implicitly, a $\mathcal{P}$-cut $\mathcal{C}_p$ of an execution $(E, \prec)$ is a poset $(\mathcal{C}_p, \prec')$; we say that $\prec'$ is the order induced by the execution on the $\mathcal{P}$-cut:

$$(e, e' \in_{\mathcal{P}} \mathcal{C}_p \wedge e \prec e') \Leftrightarrow (e \prec' e' \vee \mathcal{P}(e) \prec' e' \vee e \prec' \mathcal{P}(e') \vee \mathcal{P}(e) \prec' \mathcal{P}(e'))$$

Definition 11 ($\mathcal{P}$-consistency) *A $\mathcal{P}$-cut $\mathcal{C}_p$ is a $\mathcal{P}$-consistent cut of an execution $(E, \prec)$ if and only if:*

$$\forall e, e',\ (e \in_{\mathcal{P}} \mathcal{C}_p), e' \prec e \Rightarrow \big(e' \in_{\mathcal{P}} \mathcal{C}_p \vee Det(e'))\big)$$

Let $\mathcal{C}_p^{\sqcup}$ be the least consistent cut of $(E, \prec)$ containing all the events e such that $e \in_{\mathcal{P}} \mathcal{C}_p$; and $\mathcal{C}_p^{\sqcap}$ the greatest consistent cut smaller than $\mathcal{C}_p$. As consistent cuts form a lattice, those two cuts are well defined.

Property 3.2 $\mathcal{C}_p^{\sqcup} = \{e | \exists e' \in_{\mathcal{P}} \mathcal{C}_p,\ e \preceq e'\}$ $\qquad e \in \mathcal{C}_p^{\sqcup} \wedge \neg Det(e) \Rightarrow e \in_{\mathcal{P}} \mathcal{C}_p$ $\qquad \mathcal{C}_p^{\sqcap} \xrightarrow{*} \mathcal{C}_p^{\sqcup}$

3.3 Reduction of $\mathcal{P}$-consistent Cuts

This section defines a reduction on $\mathcal{P}$-consistent cuts and proves its correctness. This reduction relies on a global view of the system: in this section, it is always possible to know $\mathcal{C}_p^{\sqcap}$.

Definition 12 (Reduction of $\mathcal{P}$-consistent cuts) *Let $(\mathcal{C}_p, \prec)$ be a $\mathcal{P}$-consistent cut of an execution $(E, \prec^0)$. We denote $\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p$ the reduction of an event $e \notin \mathcal{C}_p$ on a $\mathcal{P}$-consistent cut $\mathcal{C}_p$. $(\mathcal{C}'_p, \prec')$ is the resulting poset defined as follows:*

- Event with a matching promised: *If $\neg Det(e) \wedge \exists \mathcal{P}(e') \in \mathcal{C}_p$ such that $\mathcal{P}(e') \lhd e$, then $\mathcal{C}'_p$ is the $\mathcal{P}$-cut $\mathcal{C}_p$ where $\mathcal{P}(e')$ is replaced by e. In particular, e takes the position that was holding $\mathcal{P}(e')$.*

$$\mathcal{C}'_p = \mathcal{C}_p\{\mathcal{P}(e') \leftarrow e'\} \text{ with } e_{\mathcal{P}} \prec' e'_{\mathcal{P}} \text{ iff } \begin{cases} e_{\mathcal{P}} \neq e' \wedge e'_{\mathcal{P}} \neq e' \wedge e_{\mathcal{P}} \prec e'_{\mathcal{P}} & \vee \\ e_{\mathcal{P}} = e' \wedge \mathcal{P}(e') \prec e'_{\mathcal{P}} & \vee \\ e'_{\mathcal{P}} = e' \wedge e_{\mathcal{P}} \prec \mathcal{P}(e') \end{cases}$$

- Deterministic event: *If* $\mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}' \wedge Det(e)$ *then* e *is inserted after* $\mathcal{C}_p^{\sqcap}$.

$$\mathcal{C}'_p = \mathcal{C}_p \cup \{e\} \text{ with } e_{\mathcal{P}} \prec' e'_{\mathcal{P}} \text{ iff } \begin{cases} e_{\mathcal{P}}, e'_{\mathcal{P}} \in \mathcal{C}_p \wedge e_{\mathcal{P}} \prec e'_{\mathcal{P}} & \vee \\ \exists e''_{\mathcal{P}} \in \mathcal{C}'_p,\ e_{\mathcal{P}} \prec' e''_{\mathcal{P}} \wedge e''_{\mathcal{P}} \prec' e'_{\mathcal{P}} & \vee \\ e'_{\mathcal{P}} = e \wedge e_{\mathcal{P}} \in \mathcal{C}_p^{\sqcap} \wedge \neg e \bowtie e_{\mathcal{P}} & \vee \\ e_{\mathcal{P}} = e \wedge e'_{\mathcal{P}} \notin \mathcal{C}_p^{\sqcap} \wedge \neg e \bowtie e'_{\mathcal{P}} \end{cases}$$

- Non-deterministic event not promised: *If* $\mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}' \wedge \neg Det(e) \wedge \forall \mathcal{P}(e') \in \mathcal{C}_p$ *such that* $\mathcal{P}(e') \not\lhd e$ *then* e *is added at the end of the* $\mathcal{P}$*-cut.*

$$\mathcal{C}'_p = \mathcal{C}_p \cup \{e\} \text{ with } e_{\mathcal{P}} \prec' e'_{\mathcal{P}} \text{ iff } \begin{cases} e_{\mathcal{P}}, e'_{\mathcal{P}} \in \mathcal{C}_p \wedge e_{\mathcal{P}} \prec e'_{\mathcal{P}} & \vee \\ \exists e''_{\mathcal{P}} \in \mathcal{C}'_p,\ e_{\mathcal{P}} \prec' e''_{\mathcal{P}} \wedge e''_{\mathcal{P}} \prec' e'_{\mathcal{P}} & \vee \\ e'_{\mathcal{P}} = e \wedge \neg e \bowtie e_{\mathcal{P}} \end{cases}$$

Property 3.3 (Reduction maintains $\mathcal{P}$-consistency) *There is a possible execution* $(E', \prec')$ *such that:*

$\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p \wedge \mathcal{C}_p$ *is a* $\mathcal{P}$*-consistent cut of* $(E, \prec) \Rightarrow \exists (E', \prec')$, $\mathcal{C}'_p$ *is a* $\mathcal{P}$*-consistent cut of* $(E', \prec')$

Wait-by-necessity is expressed in Definition 12 by the fact that only reductions on $\mathcal{C}_p^{\sqcap}$ can occur; executing the consequence of an event that did not happen is thus impossible. However, the consequence of an event that did not happen can already belong to $\mathcal{C}_p$ because it occurred in the first execution. The following property ensures that a reduction is always possible on a $\mathcal{P}$-consistent cut until $\mathcal{C}_p^{\sqcup}$, ensuring that the reduction cannot be stuck up to a consistent cut of the first execution.

Property 3.4 (no dead-lock) *A possible execution leads* $\mathcal{C}_p$ *to* $\mathcal{C}_p^{\sqcup}$*:* $\mathcal{C}_p \xrightarrow{*} \mathcal{C}_p^{\sqcup}$

This means that there is a reduction replacing all the promised events by the corresponding events. Next Section shows through theorems 3.1 and 3.2 that all the possible executions are equivalent up to $\mathcal{C}_p^{\sqcup}$ and thus all the possible executions finally provide an event corresponding to each promised event of the $\mathcal{P}$-consistent cut. In other words, we show that no dead-lock due to the wait-by-necessity synchronization could occur during the re-execution.

3.4 Recovery Cuts

We show in this section that any execution from any $\mathcal{P}$-consistent cut $\mathcal{C}_p$ eventually reaches a consistent cut of the first execution corresponding to $\mathcal{C}_p$, denoted $\mathcal{C}_H$. We first suppose here that we can identify the greatest lower consistent cut of a $\mathcal{P}$-consistent one. This cut provides the set of executable event; provided we can force only those events to be executed, the global synchronization between processes during recovery is ensured. Next section shows

that the local synchronization due to the wait-by-necessity mechanism is actually sufficient to ensure such a global synchronization.

From any cut $\mathcal{C}$, with a consistent cut $\mathcal{C}_H$ after $\mathcal{C}$, we build a $\mathcal{P}$-consistent cut $\mathcal{C}_p$ and prove that the reduction from $\mathcal{C}_p$ is constrained to reach $\mathcal{C}_H$.

Definition 13 (Recovery cut) *Let $\mathcal{C}$ be a cut of an execution $(E, \prec)$, suppose we have $\mathcal{C}_H$, a consistent cut of an $(E, \prec)$ such that $\mathcal{C} \subseteq \mathcal{C}_H$. Then we define $\mathcal{C}_p$ as follows:*

$$\mathcal{C}_p = \{e \in \mathcal{C} | Det(e) \vee \forall e', e' \prec e \Rightarrow e' \in \mathcal{C}\} \cup \{\mathcal{P}(e) | e \in \mathcal{C}_H \wedge \neg Det(e) \wedge \exists e', (e' \prec e \wedge e' \notin \mathcal{C})\}$$

All the deterministic events of $\mathcal{C}$ belong to $\mathcal{C}_p$, and thus $\mathcal{C}_p$ is obtained from $\mathcal{C}$ only by replacing some non-deterministic events by promised ones and by adding promised events.

Property 3.5 *$\mathcal{C}_p$ is a $\mathcal{P}$-consistent cut and $\mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_H$, where $\xrightarrow{*}$ only performs deterministic events.*

The following theorem states that $\mathcal{C}_H$ is also a consistent cut of the re-execution from the recovery cut, allowing us to state that the re-execution from the recovery cut is equivalent to the first execution up to $\mathcal{C}_H$.

Theorem 3.1 (Constrained execution) $\mathcal{C}_p \xrightarrow{*} \mathcal{C}'_p \Rightarrow \exists \mathcal{C}', \mathcal{C}_H \xrightarrow{*} \mathcal{C}' \wedge \mathcal{C}'_p \xrightarrow{*} \mathcal{C}'$

The following theorem states that all the possible execution from the consistent cut $\mathcal{C}_H$ can be reached by the re-execution from the recovery cut, allowing us to state thanks to theorem 3.1 that *every* re-execution from the recovery cut is equivalent to the first execution.

Theorem 3.2 (Completeness) $\mathcal{C}_H \xrightarrow{*} \mathcal{C}' \Rightarrow \mathcal{C}_p \xrightarrow{*} \mathcal{C}'$

This theorem is a direct consequence of the absence of dead-lock (Property 3.4), and shows that the forced execution can potentially reach all the existing executions passing by $\mathcal{C}_H$.

3.5 Effective Recovery

Definition 13 defines a valid recovery cut; but, the following cut $\mathcal{C}_p^2$ is also a valid recovery cut: $\mathcal{C}_p^2 = \mathcal{C} \cup \{\mathcal{P}(e) | e \in \mathcal{C}_H \backslash \mathcal{C} \wedge \neg Det(e)\}$ (proofs of Section 3.4 would be identical for $\mathcal{C}_p^2$). More generally, every cut between $\mathcal{C}_p$ and $\mathcal{C}_p^2$ (turning enough events into promised event) are also valid recovery cuts. This section defines a local reduction of a $\mathcal{P}$-consistent cut, fulfilling practical constraints and, based on this local reduction, identifies among all $\mathcal{P}$-consistent cuts that can be built from a given non-consistent cut, the one which allows to deal locally with global synchronization issues, namely the *effective recovery cut*. We only assume the following reasonable and frequently verified hypothesis:

- Reception events are non-deterministic (for simplicity).

- Reduction is defined only on local events ($s_i \xrightarrow{e} s'_i$), with a mechanism for communication: a message is automatically received some time after having been sent, or $s_i \xrightarrow{e} s'_i \wedge (e, e') \in \Gamma \Rightarrow s_j \xrightarrow{e'} s'_j$, with $s_j \in S''$, $S' \xrightarrow{*} S''$. There might be constraints on the receiving state S'' depending on the causal ordering of messages.
- The system verifies the piecewise-determinism property (Property 3.1).

Let $s_i \xrightarrow{e} s'_i$ be a reduction that happens by *uncontrolled* local execution. On the contrary, let $\exists s'_i, s_i \xrightarrow{e} s'_i$ be true if the execution of the event e can be forced from the state s_i. We then need to identify events for which execution can be forced, or *triggerable* events; we define the predicate $Trig(e)$ which is true if e is a triggerable event, depending on its kind. In practice, a message reception is a triggerable event: the message can be logged and delivered to the application when needed. On the contrary, a random choice of a number is not triggerable: it can only occur by uncontrolled execution; however, the chosen value can be controlled.

Definition 14 (Local reduction of $\mathcal{P}$-consistent cuts) *Let $(c_i, \prec_i)$ be a local $\mathcal{P}$-consistent cut of $\mathcal{C}_p$, that is to say $c_i = \{e | e \in \mathcal{C}_p \wedge e \in P_i\}$ ordered by a partial order $\prec_i$, which is the restriction of the order of $\mathcal{C}_p$ to c_i. Let the cut $cut(s_i)$ be the set of events that have already been executed to obtain s_i. Note that $cut(s_i) \subseteq c_i$. We denote $(s_i, c_i) \xrightarrow{e} (s'_i, c'_i)$ the local reduction of an event $e \notin c_i$ on a pair (local state, local $\mathcal{P}$-consistent cut). The resulting pair (s'_i, c'_i) and the poset $(c_i, \prec_i)$ are defined as follows:*

1. *If $Det(e) \wedge s_i \xrightarrow{e} s'_i \wedge \nexists \mathcal{P}(e') \in c_i$, $\mathcal{P}(e') \prec'_i e$ then the event is executed without control:*

$$(s_i, c_i) \xrightarrow{e} (s'_i, c'_i) \text{ where}$$

$$c'_i = c_i \cup \{e\} \text{ with } e_{\mathcal{P}} \prec'_i e'_{\mathcal{P}} \text{ iff } \begin{cases} e_{\mathcal{P}}, e'_{\mathcal{P}} \in c_i \wedge e_{\mathcal{P}} \prec_i e'_{\mathcal{P}} & \vee \\ \exists e''_{\mathcal{P}} \in c'_i,\ e_{\mathcal{P}} \prec'_i e''_{\mathcal{P}} \wedge e''_{\mathcal{P}} \prec'_i e'_{\mathcal{P}} & \vee \\ e'_{\mathcal{P}} = e \wedge e \in_{\mathcal{P}} c_i \wedge e \in cut(s_i) \wedge \neg e \bowtie e_{\mathcal{P}} & \vee \\ e_{\mathcal{P}} = e \wedge e \in_{\mathcal{P}} c_i \wedge e \notin cut(s_i) \wedge \neg e \bowtie e'_{\mathcal{P}} \end{cases}$$

2. *If $\neg Det(e) \wedge Trig(e) \wedge s_i \xrightarrow{e} s'_i \wedge \exists \mathcal{P}(e') \in c_i$ such that $\mathcal{P}(e') \lhd e$, then the event takes the position of its corresponding promised event but is not executed ($\prec_i$ is unchanged):*

$$(s_i, c_i) \xrightarrow{e} (s_i, c_i\{\mathcal{P}(e') \leftarrow e'\})$$

3. *If $\neg Det(e) \wedge Trig(e) \wedge s_i \xrightarrow{e} s'_i \wedge \nexists \mathcal{P}(e') \lhd e$, then the event is postponed at the end of c_i and is not executed:*

$$(s_i, c_i) \xrightarrow{e} (s_i, c'_i) \text{ where}$$

$$c'_i = c_i \cup \{e\} \text{ with } e_{\mathcal{P}} \prec'_i e'_{\mathcal{P}} \text{ iff } \begin{cases} e_{\mathcal{P}}, e'_{\mathcal{P}} \in c_i \wedge e_{\mathcal{P}} \prec_i e'_{\mathcal{P}} & \vee \\ \exists e''_{\mathcal{P}} \in c'_i,\ e_{\mathcal{P}} \prec'_i e''_{\mathcal{P}} \wedge e''_{\mathcal{P}} \prec'_i e'_{\mathcal{P}} & \vee \\ e'_{\mathcal{P}} = e \wedge \neg e \bowtie e_{\mathcal{P}} \end{cases}$$

4. *If* $\exists s'_i, s_i \xrightarrow{e} s'_i \wedge e$ *minimal in* $c_i \backslash cut(s_i)$ *then the event is artificially triggered (*$\prec_i$ *is unchanged):*

$$(s_i, c_i) \xrightarrow{e} (s'_i, c_i)$$

5. *If* $s_i \xrightarrow{e} s'_i \wedge \neg Det(e) \wedge \neg Trig(e) \wedge \exists \mathcal{P}(e')$ *minimal in* $c_i \backslash cut(s_i)$ *s.t.* $\mathcal{P}(e') \lhd e$*, and* s''_i *is such that* $s_i \xrightarrow{e'} s''_i$*, then there is a promised event that can force* e *to happen* as in the first execution*, provided this event should be executed now (*$\prec_i$ *is unchanged):*

$$(s_i, c_i) \xrightarrow{e'} (s''_i, c_i\{\mathcal{P}(e') \leftarrow e'\})$$

6. *If* $s_i \xrightarrow{e} s'_i \wedge \neg Det(e) \wedge \neg Trig(e) \wedge \nexists \mathcal{P}(e') \in c_i, \mathcal{P}(e') \lhd e \wedge \forall e'_{\mathcal{P}} \in c_i \backslash cut(s_i)$, $e'_{\mathcal{P}} \bowtie e$*, then there is no more conflicting promised event; the event can be executed as it is.*

$$(s_i, c_i) \xrightarrow{e} (s'_i, c'_i) \text{ where}$$

$$c'_i = c_i \cup \{e\} \text{ with } e_{\mathcal{P}} \prec'_i e'_{\mathcal{P}} \text{ iff } \begin{cases} e_{\mathcal{P}}, e'_{\mathcal{P}} \in c_i \wedge e_{\mathcal{P}} \prec_i e'_{\mathcal{P}} & \vee \\ \exists e''_{\mathcal{P}} \in c'_i,\ e_{\mathcal{P}} \prec'_i e''_{\mathcal{P}} \wedge e''_{\mathcal{P}} \prec'_i e'_{\mathcal{P}} & \vee \\ e'_{\mathcal{P}} = e \wedge \neg e \bowtie e_{\mathcal{P}} \end{cases}$$

This local reduction must ensure that only reductions defined by Definition 12 are accepted.

Theorem 3.3 (Constrained execution and completeness)

$$(s_i, c_i) \xrightarrow{*} (s'_i, c'_i) \Rightarrow \exists (s''_i, c''_i), \mathcal{C}, \mathcal{C}_H \xrightarrow{*} \mathcal{C} \wedge c''_i = \{e | e \in \mathcal{C} \wedge e \in P_i\} \wedge cut(s''_i) = c''_i$$

$$\mathcal{C}_H \xrightarrow{*} \mathcal{C} \Rightarrow (s_i, c_i) \xrightarrow{*} (s'_i, c'_i) \wedge c'_i = \{e | e \in \mathcal{C} \wedge e \in P_i\} \wedge cut(s'_i) = c'_i$$

Reduction of $\mathcal{P}$-consistent cuts relies on on the reduction $\mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}'_p$. The objective here is to prevent events that could not be executed from $\mathcal{C}_p^{\sqcap}$ to be executed, but also to ensure that all the events that could be executed from $\mathcal{C}_p^{\sqcap}$ are executed. First, the local consistency of a $\mathcal{P}$-cut (definition 10) ensures that all the deterministic events that locally occur before a deterministic event belong to the $\mathcal{P}$-cut. Secondly, the reduction defined above ensures that all the local non-deterministic events of the causal past of an event occur before this event; the local synchronization issue is then tackled.

We now focus on the global synchronization issues due to messages that cross the recovery cut.

Orphan Messages A message is said to be orphan relatively to a cut if the message reception belongs to the cut but not the message sending. Thus, the consequence of the reception must not be executed before the message has been sent again. If, by construction, a promised event correspond to each orphan reception, then the local wait-by-necessity mechanism ensures that the local consequence of an orphan reception is not executed before its corresponding sending; there is thus no need for global synchronization between events.

In Transit Messages A message is said to be in-transit relatively to a cut if the message sending belongs to the cut but not the message reception. The re-execution cannot automatically re-send in-transit messages; the communication reduction rule cannot be supposed anymore because the sending of the message has not occurred in the same execution. In-transit messages must then be logged *with the corresponding application data* in order to replay their reception upon re-execution.

As causal relations between events that occur by re-execution and events that are replayed from a log are lost, the message ordering is thus not ensured in the case of in-transit messages. As a consequence, a promised event must correspond to each reception of in-transit message; synchronization is ensured as for orphan receptions. In other words, $\mathcal{C}_H$ must contains all the receptions of in-transit messages.

To conclude, the recovery cut $\mathcal{C}_p$ must be built as follows:

Effective recovery cut construction To be able to recover an application from a non-consistent global state $\mathcal{C}$, the corresponding recovery cut $\mathcal{C}_p$ must be built as follows:

$$\mathcal{C}_p = \mathcal{C}\{e \leftarrow \mathcal{P}(e) | \exists e', (e', e) \in \Gamma \wedge e' \notin \mathcal{C}\} \cup \{\mathcal{P}(e) | e \in \mathcal{C}_H \backslash \mathcal{C} \wedge \neg Det(e)\}$$

with $\forall e, (e, e') \in \Gamma \wedge e' \notin \mathcal{C} \Rightarrow \mathcal{P}(e) \in \mathcal{C}_H$ and e is logged.

A recovery cut is then built in two steps. First, a global state $\mathcal{C}$ is stored on a stable storage. It does not require any synchronization between processes; each process checkpoints independently. Once there are no more messages in transit[1] in $\mathcal{C}$, each process adds to its checkpoint c_i, its local part of $\mathcal{C}_p$, which is an ordered set of promised events. It also adds the content of in-transit messages it received since its checkpoint. Only when all processes have completed that second step, the global state $\mathcal{C}$ is recoverable.

4 A Fault-Tolerance protocol for ASP

This section presents a fault-tolerance protocol for the ASP calculus [8, 3]. This protocol has been implemented in ProActive, a Java implementation of the ASP calculus [2]. $\mathcal{P}$-consistency is necessary because the constrained checkpointability in ProActive does not permit the creation of consistent global states. We show $\mathcal{P}$-consistency and thus recoverability of the global states built by the protocol.

4.1 ASP: a Model of Distributed Objects

The ASP calculus considers piecewise-deterministic mono-threaded activities (processes) communicating with an asynchronous request and reply mechanism. Sending a request from an activity i to another activity j consists in adding an entry in the pending requests queue of the activity j. Requests are then *served* by the destination activity: the destination

[1] The moment when all in-transit messages have been received can be identified by for example sending specifics messages to empty out communication channels.

activity reacts to this request and associates a result to be returned to the sender. ASP is called asynchronous because requests can be served by j later than their arrival and, in the meanwhile, i can continue its execution the result of the request is not needed. Asynchrony is achieved by creating a *future* when a request is sent; a future is a place-holder for a not-yet-received result. If an activity tries to access a future that is still only a placeholder, then this activity is blocked until the reply corresponding to this future is received (wait-by-necessity).

In this paper, we use two main properties of ASP model, proved in [8]:

Replies can be sent in any order without any consequence on the execution.

In particular, [8] proves that reply events do not conflict with other events, and thus *replies receptions are deterministic events*. Moreover,

An execution is fully characterized by the order of request senders on each activity.

In other words, the only events that conflict are request receptions originating from different activities and arriving at the same destination. Thus only request receptions are non-deterministic.

4.2 A Potential Causality for ASP

The asynchronous service of requests introduces particular relations between events on a single process: a request reception is not causally related with an internal event until the request is served, in the meanwhile it is only related with other request receptions. Moreover, a reply reception is only causally related with the first use of the associated future (and with the sending of this reply). We then define here a potential causality relation for ASP that takes into account those particularities. For a given ASP execution $S \xrightarrow{e^1 \dots e^n} S'$, let $\Gamma_Q \subseteq \{(e, e') \in \{e^1 \dots e^n\} * \{e^1 \dots e^n\}\}$ be the set of pairs sending-reception of requests, $\Gamma_R \subseteq \{(e, e') \in \{e^1 \dots e^n\} * \{e^1 \dots e^n\}\}$ be the set of pairs sending-reception of replies, and $\Sigma \subseteq \{(e, e') \in \{e^1 \dots e^n\} * \{e^1 \dots e^n\}\}$ be the set of pairs reception-service of requests. Finally, we denote $\Gamma = \Gamma_Q \cup \Gamma_R$. $\prec^{ASP}$ is a convenient local causal order for ASP:

Definition 15 (Local order) *$\prec_i^{ASP}$ is a* partial *order defined by: $e \prec_i^{ASP} e'$ if and only if*

$$\begin{cases} (\nexists e^k, [(e^k, e) \in \Gamma \vee (e^k, e') \in \Gamma] \wedge e \prec_i^{hb} e') \vee & (a)\ \textit{internal events totally ordered} \\ (\exists e^k, e^l, [(e^k, e) \in \Gamma_Q \wedge (e^l, e') \in \Gamma_Q \wedge e \prec_i^{hb} e') \vee & (b)\ \textit{request receptions totally ordered} \\ ((e, e') \in \Sigma \vee & (c)\ \textit{service causality} \\ (\exists e^k, e \prec_i^{ASP} e^k \prec_i^{ASP} e') & (c)\ \textit{transitivity} \end{cases}$$

Definition 16 (Potential causality for ASP) $\prec^{ASP}$ *is a* potential causality *order for ASP, defined as follows:* $e \prec^{ASP} e'$ *if and only if*

$$\begin{cases} e \prec_i^{ASP} e' \vee & (a) \quad \textit{local ordering} \\ ((e,e') \in \Gamma \vee & (c) \quad \textit{communication causality} \\ (\exists e^k, e \prec^{ASP} e^k \prec^{ASP} e') & (c) \quad \textit{transitivity} \end{cases}$$

Moreover, ASP ensures a causal ordering of requests:

$$(e,e_1) \in \Gamma_Q \wedge (e',e_2) \in \Gamma_Q \wedge e \prec^{ASP} e' \wedge e_1 \text{ and } e_2 \text{ occur on the same process} \Rightarrow e_1 \prec^{ASP} e_2$$

As we have shown in Section 2.2, the total local ordering of message receptions is sufficient to ensure that causal ordering will be ensured at re-execution, provided in-transit messages are all logged and correspond to a promised event (Section 3.5).

4.3 Promised Events in ASP

In ASP, the only non-deterministic events are request receptions. Promised events for request receptions are called *promised request*. A promised request can be placed in the request queue of an activity as a standard request. Due to the wait-by-necessity property, a promised request cannot be served: an activity that should serve a promised request is blocked until the corresponding request is received. Recall that this ensures the causal order: requests ordered regarding other request receptions (rule (b)), and has not to be ordered with other events since a promised request cannot be served (ensuring rule (c)).

Promised requests are totally ordered between themselves. As the execution is only characterized by the order of activities sending requests, promised requests only need to log the identifier of the request sender to enforce the same event to occur again at re-execution (point-to-point FIFO communication).

4.4 A Fault-Tolerance Protocol for ASP

The proposed protocol is an index-based Communication-Induced-Checkpointing protocol based on [7]. The idea is to piggyback checkpoints logical clocks on the messages of the application; on message reception, a process schedules a checkpoint *to the next checkpointable state, or stable state* if the piggybacked clock is greater than the local clock. As a consequence, the constituted global states cannot be *consistent*. The piggybacked clock allows to identify orphan and in-transit messages. $\mathcal{P}$-consistent global state can then be built: orphan events between the scheduling of a checkpoint and the checkpoint itself and all the non-deterministic events between the checkpoint and the termination of the global state are turned promised. As a stable state is always reached before any request service, an orphan promised request is *never served before the next checkpoint*. More details about the protocol and the notion of stable states can be found in [10] and [2].

Figure 1 shows a simplified version of our protocol than can create one $\mathcal{P}$-consistent global state. Let $scheduleCkpt_i$ be a local variable such that the procedure **Checkpoint** is called if

it is true and a stable state is reached; we suppose here that at least one $scheduleCkpt_i$ is set to true automatically. The procedure **GlobalStateTermination** is called on at least one activity when the global state is completed. Let $status_i \in \{running, checkpointed, closed\}$ be a local variable piggybacked on each message from i to j; thanks to this variable, we can define two predicates $inTransit(M_{i,j})$ and $orphan(M_{i,j})$ that indicate if the message $M_{i,j}$ is orphan or in transit. $\mathbb{H}_i$ is a list that contains all the promised events that occurs after the checkpoint and log_i a list that contains the in-transit messages; those lists can be added to the checkpoint C_i of the activity i.

- **Reception of $M_{i,j}$ on j**
  if $(status_i \neq running)$ then $scheduleCkpt_j = true$ if $(status_i = closed)$ then
    $status_j = closed$
    and $C_j.add(\mathbb{H}_j)$
    and $C_j.add(log_j)$
  if $(status_j = checkpointed)$ then $\mathbb{H}_j.append(\mathcal{P}(M_{i,j}))$
  if $(inTransit(M_{i,j}))$ then $log_j.append(M_{i,j})$

- **GlobalStateTermination on i**
  $status_i = closed$
  $C_i.add(\mathbb{H}_i)$
  $C_i.add(log_i)$

- **Checkpoint on i**
  $\forall M_{j,i} \in C_i$, if $(orphan(M_{j,i}))$ then $M_{j,i} \rightarrow \mathcal{P}(M_{j,i})$
  $status_i = checkpointed$

Figure 1: The simplified checkpointing protocol for ASP

The correctness of the protocol is ensured by the fact that the recovery state we build fulfills the hypothesis required in Section 3. The recovery cut is formed by checkpoints in which orphan requests have been promised; moreover a promised request is added for all the request receptions until the global state termination, which ensures that every in-transit messages has a promised event. The consistency of $\mathcal{C}_H$ is ensured by the closure mechanism of the *closed* state. All in-transit messages are logged.

Concerning synchronization, a wait-by-necessity automatically occurs upon the service of a promised request, which is sufficient provided no orphan request are served before the checkpoint. For results (future values), we chose a safe approximation of the synchronization by waiting that all the futures that had been received before the checkpoint are received before restarting the activity.

5 Conclusion

Not just taking the happened-before as the causal link, this paper relaxes causality between events and provides a formal treatment for potential causality. Potential causality is parameterized by a semantics, taking into account equivalent executions. We defined the concept of *Promised consistency*. It captures a recoverability condition on a global state, putting at work the idea of a *promised event*: a place holder for a non-deterministic event, subject to the necessary synchronization upon recovery. Using promises, no systematic logging of application data is needed: promises are filled up with the data produced by re-execution. A recovery starts with the identification of a $\mathcal{P}$-consistent state to restart from, promises force the re-execution to reach an actual global state of the first execution. We finally identified a

local reduction that allows to bypass any global synchronization upon recovery from a well-defined *effective recovery cut*. Finally, this formalism was applied to show the $\mathcal{P}$-consistency of the global states created by an existing fault-tolerance protocol.

References

[1] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[2] Francoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. A hybrid message logging-cic protocol for constrained checkpointability. In *Proceedings of EuroPar2005*, number 3648 in LNCS, pages 644–653, Lisbon, Portugal, August-September 2005. Springer.

[3] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer-Verlag New York, Inc., 2005.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Transactions on Computer Systems*, pages 63–75, 1985.

[5] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communications. *Distributed Computing*, 9(4):173–191, 1996.

[6] Om P. Damani, Ashis Tarafdar, and Vijay K. Garg. Optimistic recovery in multi-threaded distributed systems. In *Symposium on Reliable Distributed Systems*, pages 234–243, 1999.

[7] D.Briatico, A.Ciuffoletti, and L.Simoncini. A distributed domino-effect free recovery algorithm. In *IEEE International Symposium on Reliability, Distributed Software, and Databases*, pages 207–215, 1984.

[8] D.Caromel, L.Henrio, and B.Serpette. Asynchronous and deterministic objects. In *31st ACM Symposium on Principles of Programming Languages*. ACM Press, 2004.

[9] M. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, oct 1996.

[10] F.Baude, D.Caromel, C.Delbé, and L.Henrio. A fault tolerance protocol for asp calculus : Design and proof. Technical Report RR-5246, INRIA, 2004.

[11] G.Bronevetsky, D.Marques, K.Pingali, and P.Stodghill. Automated application-level checkpointing of mpi programs. *SIGPLAN Not.*, 38(10):84–94, 2003.

[12] Jean-Michel Hélary, Robert H. B. Netzer, and Michel Raynal. Consistency issues in distributed checkpoints. *IEEE Trans. Softw. Eng.*, 25(2):274–281, 1999.

[13] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proc. 7th Annual ACM Symp. on Principles of Distributed Computing*, pages 171–181, Toronto (Canada), 1988.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, volume 21, pages 558–565, July 1978.

[15] Hong Va Leong and Divyakant Agrawal. Using message semantics to reduce rollback in optimistic message logging recovery schemes. In *International Conference on Distributed Computing Systems*, pages 227–234, 1994.

[16] D. Manivannan and Mukesh Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):703–713, 1999.

[17] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel And Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[18] R.E.Strom and S.Yemini. Optimistic recovery in distributed systems. In *ACM Transactions on Computer Systems*, volume 3, pages 204–226, 1985.

[19] Luis Moura Silva and Joao Gabriel Silva. Using message semantics for fast-output commit in checkpointing-and-rollback recovery, 1999.

[20] Ashis Tarafdar and V Garg. Happened before is the wrong model for potential causality. Technical Report TR-PDS-1998-006, Parallel & Dsitributed Systems Group, University of Texas, july 1998.

[21] Ashis Tarafdar and Vijay K. Garg. Adressing false causality while detecting predicates in distributed programs. In *18th International Conference on Distributed Computing Systems*, pages 94–101, May 1998.

[22] Nitin H. Vaidya. Staggered consistent checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):694–702, 1999.

Appendix

This appendix contains the proofs of most of the properties and theorems presented in this paper. The last section focuses on the technical details of Section 3.5; it also presents properties that were not mentioned in the paper but are necessary to prove Theorem 3.3.

A Property 3.2: Properties of $\mathcal{C}_p^{\sqcup}$ and $\mathcal{C}_p^{\sqcap}$

$$\mathcal{C}_p^{\sqcup} = \{e | \exists e' \in_{\mathcal{P}} \mathcal{C}_p,\ e \preceq e'\} \qquad e \in \mathcal{C}_p^{\sqcup} \wedge \neg Det(e) \Rightarrow e \in_{\mathcal{P}} \mathcal{C}_p \qquad \mathcal{C}_p^{\sqcap} \xrightarrow{*} \mathcal{C}_p^{\sqcup}$$

Proof : $\{e | \exists e' \in_{\mathcal{P}} \mathcal{C}_p,\ e \preceq e'\}$ is trivially a consistent cut, and contains all the events $e \in_{\mathcal{P}} \mathcal{C}_p$; minimality is trivial. Let $e \in \mathcal{C}_p^{\sqcup}$, and $\neg Det(e)$. Thus $\exists e' \in_{\mathcal{P}} \mathcal{C}_p$, $e \preceq e'$, so by Definition 11 $e \in_{\mathcal{P}}$. □

B Property 3.3: Reduction maintains $\mathcal{P}$-consistency

There is a possible execution $(E', \prec')$ such that:

$$\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p \wedge \mathcal{C}_p \text{ is a } \mathcal{P}\text{-consistent cut of } (E, \prec) \Rightarrow \mathcal{C}'_p \text{ is a } \mathcal{P}\text{-consistent cut of } (E', \prec')$$

Proof : This proof focuses on non-trivial cases: cases involving transitive closure of causal order relations are skipped, they would be proved by a recurrence on the length of the inference necessary to assert $e \prec e'$. Moreover, we focus on the arguments relative to the newly added event e.

- If $Det(e)$, $\mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}'$, then there is an execution equivalent to the one that generated E such that $S \xrightarrow{*} \mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}'$ ($\mathcal{C}_p^{\sqcap}$ is a cut of E and e is minimal of $(E, \prec^0) \backslash \mathcal{C}_p^{\sqcap}$).$\mathcal{C}'$ is consistent.

  We first prove that $\mathcal{C}'_p$ is a P-cut. As $Det(e)$, suppose $e' \prec_i^0 e$, then $e' \in \mathcal{C}' \Rightarrow e' \in \mathcal{C}'_p$ ($\mathcal{C}'$ is consistent and $\mathcal{C}' \subseteq \mathcal{C}'_p$ by construction). If $e \prec_i^0 e'$ with $e' \in \mathcal{C}'_p$, then first $\neg Det(e')$ (else $\mathcal{C}_p$ would not be a P-cut), and $Det(e)$ is sufficient to conclude that $\mathcal{C}'_p$ is a P-cut.

  Second, we prove that $\prec'$ is the order induced by the potential order $\prec^0$ on $\mathcal{C}'_p$. Suppose $e' \prec^0 e$ then $e' \in \mathcal{C}' \backslash \{e\} = \mathcal{C}_p^{\sqcap}$ then $e' \in \mathcal{C}_p$, and $e' \prec' e$ or $e' \bowtie e$ by definition of the reduction, as $e' \prec^0 e \Rightarrow \neg e' \bowtie e$ (Definition 6), we have $e' \prec' e$. Suppose $e \prec^0 e'$ then $e' \notin \mathcal{C}_p^{\sqcap}$; thus, $e \in_{\mathcal{P}} \mathcal{C}_p$ implies $e \prec' e'$, and $\prec^0 \Rightarrow \prec'$. Reciprocally, suppose $e' \prec' e$, then $e' \in \mathcal{C}_p^{\sqcap}$, so there is an execution belonging to $(E, \prec^0)$ such that $S^0 \xrightarrow{*e'*} \mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}'$, Definition 6 implies either $e \bowtie e'$ or $e' \prec^0 e$. If $e \prec' e'$ then $e' \notin \mathcal{C}_p^{\sqcap}$, thus, as e is minimal in $E \backslash \mathcal{C}_p^{\sqcap}$ (Definition 7), one of the executions belonging to E is $\mathcal{C}_p^{\sqcap} \xrightarrow{e*e'}$ thus either $e \bowtie e'$ or $e' \prec^0 e$.

Finally, $\mathcal{C}'_p$ is a $\mathcal{P}$-consistent cut of $(E, \prec^0)$ because $e \prec^0 e' \Rightarrow Det(e)$ and $e' \prec^0 e \Rightarrow e' \in \mathcal{C}' \Rightarrow e' \in \mathcal{C}'_p$ ($\mathcal{C}'$ is consistent and $\mathcal{C}' \subseteq \mathcal{C}'_p$ by construction).

- If $\neg Det(e) \wedge \exists \mathcal{P}(e)_0 \in \mathcal{C}_p$ such that $\mathcal{P}(e_0) \lhd e$, then first, $\mathcal{C}'_p$ is a P-cut of $(E, \prec)$, Definition 10 has the same requirements for e_0 and $\mathcal{P}(e_0)$. Second, $e_0 \prec^0 e' \Leftrightarrow \mathcal{P}(e_0) \prec e' \Leftrightarrow e \prec' e'$, and the same for $e' \prec \mathcal{P}(e_0)$; so $\prec'$ is the order induced by the potential order $\prec^0$ on $\mathcal{C}'_p$ (it is exactly the same order as $\prec$ with $\mathcal{P}(e_0)$ replaced by e_0). Finally, $\mathcal{C}'_p$ is a $\mathcal{P}$-consistent cut of $(E, \prec)$, Definition 11 has the same requirements for e_0 and $\mathcal{P}(e_0)$.

- Else if $\neg Det(e) \wedge \nexists \mathcal{P}(e_0) \in \mathcal{C}_p$ such that $\mathcal{P}(e_0) \lhd e$ then Properties 3.1 and 3.2 ensure that $S \xrightarrow{*} \mathcal{C}_p^{\sqcup} \xrightarrow{e} \mathcal{C}''$, $\mathcal{C}''$ is a cut of a possible executions $(E', \prec^1)$.

  First, $\mathcal{C}'_p$ is a P-cut of $(E', \prec^1)$. Indeed, $e' \prec^1 e \Rightarrow e' \in \mathcal{C}_p^{\sqcup}$, then either $Det(e')$ or $e' \in_{\mathcal{P}} \mathcal{C}_p$ (Property 3.2), and if $e \prec^1 e'$ then $e' \notin \mathcal{C}'_p$.

  Second, $\prec'$ is the order induced by the potential order $\prec^1$ on $\mathcal{C}'_p$. Suppose $e' \prec^1 e$, with $e' \in_{\mathcal{P}} \mathcal{C}'_p$, then $e' \in_{\mathcal{P}} \mathcal{C}_p$, Definition 6 implies $\neg e' \bowtie e$ and thus $e' \prec' e$. Suppose $e \prec^1 e'$, then $e' \notin \mathcal{C}_p^{\sqcup}$ and thus $e' \notin_{\mathcal{P}} \mathcal{C}_p$ (contradictory). Reciprocally, if $e' \prec' e$ (one never has $e \prec' e'$) then $\neg e' \bowtie e$ and $e' \in \mathcal{C}_p^{\sqcup}$, Definition 6 implies $e' \prec^1 e$.

  Finally, we prove the $\mathcal{P}$-consistency. If $e' \prec^1 e$ then $e' \in \mathcal{C}_p^{\sqcup}$ and Property 3.2 implies $Det(e') \vee e' \in_{\mathcal{P}} \mathcal{C}_p \subseteq \mathcal{C}'_p$. If $e \prec^1 e'$ then $e' \notin \mathcal{C}'^{\sqcup}_p$ because $e \notin \mathcal{C}_p^{\sqcup}$ and $\mathcal{C}_p^{\sqcup}$ is consistent, thus $e' \notin_{\mathcal{P}} \mathcal{C}_p$ and $e' \notin_{\mathcal{P}} \mathcal{C}_p$. Thus, $\mathcal{C}'_p$ is a $\mathcal{P}$-consistent cut of the execution $(E', \prec'')$ (Definition 11).

□

C Property 3.4: No dead-lock

A possible execution leads $\mathcal{C}_p$ to $\mathcal{C}_p^{\sqcup}$: $\mathcal{C}_p \xrightarrow{*} \mathcal{C}_p^{\sqcup}$

Proof : We will prove that $\forall \mathcal{C}_p$ $\mathcal{P}$-consistent, if $\mathcal{C}_p$ is not consistent then $\exists \mathcal{C}'_p \exists e \in \mathcal{C}_p^{\sqcup} \backslash \mathcal{C}_p, \mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p$. Then, by recurrence, there is a way of reducing $\mathcal{C}_p$ to $\mathcal{C}_p^{\sqcup}$.

$\mathcal{C}_p^{\sqcup} \neq \mathcal{C}_p^{\sqcap}$, $\mathcal{C}_p^{\sqcup}$ and $\mathcal{C}_p^{\sqcap}$ being cuts of the same execution, $\exists e \in \mathcal{C}_p^{\sqcup} \backslash \mathcal{C}_p^{\sqcap}$, $\mathcal{C}_p^{\sqcap} \xrightarrow{e*} \mathcal{C}_p^{\sqcup}$, Moreover, e is either a deterministic event or corresponds to a *minimal* promised event of $\mathcal{C}_p$, and $\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p$. Indeed, $\exists e \in \mathcal{C}_p^{\sqcup} \backslash \mathcal{C}_p^{\sqcap}$, $\mathcal{C}_p^{\sqcap} \xrightarrow{e*} \mathcal{C}_p^{\sqcup}$ implies e is minimal in $\mathcal{C}_p^{\sqcup} \backslash \mathcal{C}_p^{\sqcap}$ (Definition 6) then $e \notin \mathcal{C}_p$ (else we should have $e \in \mathcal{C}_p^{\sqcap}$), and thus $\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p$. Moreover, if $\neg Det(e)$ then $\mathcal{P}(e) \in \mathcal{C}_p$ (Property 3.2) and $\mathcal{P}(e) \lhd e$. □

D Property 3.5

$\mathcal{C}_p$ is a $\mathcal{P}$-consistent cut and $\mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_H$, where $\xrightarrow{*}$ only performs deterministic events.

Proof : As a prerequisite, it is easy to prove that $\neg Det(e) \wedge e \in \mathcal{C}_H \Rightarrow e \in_{\mathcal{P}} \mathcal{C}_p$. Let us

first prove that $\mathcal{C}_p$ is a P-cut. Let $e \in_{\mathcal{P}} \mathcal{C}_p$, and $e' \prec_i e$. If $Det(e)$ then $e \in \mathcal{C}$, and $e' \in \mathcal{C}$ ($\mathcal{C}$ is a cut); if moreover $Det(e')$ then $e' \in \mathcal{C}_p$, else $e' \in \mathcal{C} \subseteq \mathcal{C}_H$ and $e \in_{\mathcal{P}} \mathcal{C}_p$.

$\mathcal{P}$-consistency of $\mathcal{C}_p$ is then ensured by the fact that $e \in_{\mathcal{P}} \mathcal{C}_p$, and $e' \prec e$ implies that $e \in \mathcal{C}_H$ and $e' \in \mathcal{C}_H$, thus either $Det(e')$ or $e' \in_{\mathcal{P}} \mathcal{C}_p$. $\mathcal{C}_p^{\sqcup}$ and $\mathcal{C}_H$ are consistent cuts of $(E, \prec)$ and $\mathcal{C}_p^{\sqcup} \subseteq \mathcal{C}_H$ by definition of $\mathcal{C}_p^{\sqcup}$, thus $\mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_H$ (moreover, events missing in $\mathcal{C}_H$ are all deterministic). □

E Theorem 3.1: Constrained execution

$$\mathcal{C}_p \xrightarrow{*} \mathcal{C}'_p \Rightarrow \exists \mathcal{C}', \mathcal{C}_H \xrightarrow{*} \mathcal{C}' \wedge \mathcal{C}'_p \xrightarrow{*} \mathcal{C}'$$

Proof : We will prove that:

$$\mathcal{C}_p \xrightarrow{*} \mathcal{C}'_p \Rightarrow \mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_p'^{\sqcup}$$

Indeed, Property 3.4 ensures $\mathcal{C}'_p \xrightarrow{*} \mathcal{C}_p'^{\sqcup}$, as moreover $\mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_H$ only executing deterministic events, Definition 7 ensures that there is $\mathcal{C}'$ such that $\mathcal{C}_p'^{\sqcup} \xrightarrow{*} \mathcal{C}'$ and $\mathcal{C}_H \xrightarrow{*} \mathcal{C}'$.

By recurrence on the length of the reduction $\mathcal{C}_p \xrightarrow{*} \mathcal{C}'_p$. If the length is zero then $\mathcal{C}_p = \mathcal{C}'_p$, trivial. Suppose $\mathcal{C}_p \xrightarrow{*} \mathcal{C}'_p \wedge \mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_p'^{\sqcup}$, let $\mathcal{C}'_p \xrightarrow{e} \mathcal{C}''_p$, $\mathcal{C}''_p$ is a $\mathcal{P}$-consistent cut of an execution $(E'', \prec'')$ (Property 3.3).

- If $e \in \mathcal{C}_p'^{\sqcup}$, then $Det(e) \vee \mathcal{P}(e) \in \mathcal{C}'_p$ (Property 3.2), and thus $\mathcal{C}_p'^{\sqcup} = \mathcal{C}_p''^{\sqcup}$, indeed both are consistent cuts of the same execution (if $Det(e)$ or $\mathcal{P}(e) \in \mathcal{C}'_p$ the proof of Property 3.3 ensures that the reference execution remains the same). Finally $\mathcal{C}_p^{\sqcup} \xrightarrow{*} \mathcal{C}_p''^{\sqcup}$ (recurrence hypothesis).

- If $e \notin \mathcal{C}_p'^{\sqcup}$, then there is a cut $\mathcal{C}_1$ of the execution $(E'', \prec'')$ such that $\mathcal{C}_p'^{\sqcup} \xrightarrow{e} \mathcal{C}_1$. Indeed, if $Det(e)$ then e is compatible with all the events in $\mathcal{C}_p'^{\sqcup} \backslash \mathcal{C}_p'^{\sqcap}$: $\mathcal{C}'_p \xrightarrow{e} \mathcal{C}''_p \Rightarrow \exists \mathcal{C}_2$, $\mathcal{C}_p'^{\sqcap} \xrightarrow{e} \mathcal{C}_2$, and, as $\mathcal{C}_p'^{\sqcap} \xrightarrow{*} \mathcal{C}_p'^{\sqcup}$, $\exists \mathcal{C}_1$, $\mathcal{C}_p'^{\sqcup} \xrightarrow{e} \mathcal{C}_1$ (one can choose $(E'', \prec'') = (E', \prec')$). Else $\neg Det(e)$, similarly, $\exists \mathcal{C}_2$, $\mathcal{C}_p'^{\sqcap} \xrightarrow{e} \mathcal{C}_2$, Property 3.1 ensures that $\exists \mathcal{C}_1$, $\mathcal{C}_p'^{\sqcup} \xrightarrow{e} \mathcal{C}_1$ (as no e' in $\mathcal{C}_p^{\sqcup}$ is such that $\mathcal{P}(e') \lhd e$).

  Finally $\mathcal{C}_1 = \mathcal{C}_p''^{\sqcup}$ because $\mathcal{C}''_p \subseteq \mathcal{C}$, $\mathcal{C}_1$ is consistent, and $\mathcal{C}_1$ only differs from $\mathcal{C}_p'^{\sqcup}$ by adding e (if there was a consistent cut containing $\mathcal{C}''_p$ smaller than $\mathcal{C}_1$, this would contradict the minimality of $\mathcal{C}_p'^{\sqcup}$).

□

F Technical Details for Section 3.5

We will prove in this section that the reduction presented in the proof of the no deadlock property (Property 3.4) can be performed by only applying the rule for deterministic

reduction and the two rules for non-triggerable events. Then, as non-triggerable events can be considered as a triggerable ones, one can safely consider some events as triggerable and add the three rules supposing $Trig(e)$.

We will refer to the reductions of the definition 14 as reduction **(k)** with $k \leq 6$

An additional (technical) requirement on the local execution is used in this proof:

Property F.1 *The local adaptation of determinism and piecewise-determinism is as follows*

$$s_i \xrightarrow{*} s'_i \wedge s_i \xrightarrow{e} s^1_i \wedge e \notin cut(s'_i) \wedge Det(e) \Rightarrow \exists s^2_i,\ s'_i \xrightarrow{e} s^2_i$$

$$s_i \xrightarrow{*} s'_i \wedge s_i \xrightarrow{e} s^1_i \wedge \neg Det(e) \wedge \forall e' \in cut(s'_i) \mathcal{P}(e) \not\lhd e' \Rightarrow \exists s^2_i,\ s'_i \xrightarrow{e} s^2_i$$

If $s_i \xrightarrow{e} s^1_i$ belong to the same execution as s'_i then this property relies on the definition of the local execution, which should be piecewise-deterministic by hypothesis. However, suppose $(e_s, e) \in \Gamma$, $e \in s'_i \backslash s_i$ and $s_i \xrightarrow{e} s^1_i$ belongs to a first execution, and s'_i to a second one. Then, $s_i \xrightarrow{e} s^1_i$ does not imply $\exists s^2_i$, $s_i \xrightarrow{e} s^2_i$ by nature. Thus, $\exists s^2_i$, $s_i \xrightarrow{e} s^2_i$ must be simulated by the logging of in-transit messages. Each usage of Property F.1 below supposes that in-transit messages are logged.

First, one can check easily (by recurrence on the reduction) that every element that is in c_i but not in $cut(s_i)$ is non-deterministic, moreover, at the beginning all those elements are promised, and the only way to add a non-promised in c_i but not is in $cut(s_i)$ are the reductions **(2)** and **(3)**, thus the following property:

Property F.2

$$e_{\mathcal{P}} \in c_i \backslash cut(s_i) \Rightarrow \neg Det(e) \wedge (Trig(e) \vee \exists e',\ e_{\mathcal{P}} = \mathcal{P}(e'))$$

On the other side, all events in $cut(s_i)$ belong (possibly promised) to s_i:

Property F.3

$$e \in cut(s_i) \Rightarrow e \in_{\mathcal{P}} c_i$$

In the following we denote $c_i^{\sqcap} = \{e | e \in \mathcal{C}_p^{\sqcap} \wedge e \in P_i\}$; we naturally extend the reduction on states to $c_i^{\sqcap}$, because of consistency. However, we will have to ensure that $c_i = \{e | e \in \mathcal{C}_p \wedge e \in P_i\}$, because c_i is already defined by the reductions of Definition 14.

First we state the sufficiency of rules **(1)**, **(5)**, and **(6)**.

Property F.4 *If one applies the reductions as specified in the Property 3.4 (no dead-lock), i.e. deterministic and minimal promised events first, then reductions **(1)**, **(5)**, and **(6)** are sufficient to perform the same execution:*

$\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p \wedge (Det(e) \vee e$ *minimal promised event*$) \Rightarrow (s_i, c_i) \xrightarrow{e} (s'_i, c'_i)$ *by reductions **(1)**, **(5)**, and **(6)***

And for each step of the reduction $c_i = \{e | e \in \mathcal{C}_p \wedge e \in P_i\}$ and $c_i^{\sqcap} \subseteq s_i$.

Proof : We suppose that for each step of the reduction $c_i = \{e | e \in \mathcal{C}_p \wedge e \in P_i\}$ and $\mathcal{C}_p \xrightarrow{e} \mathcal{C}'_p \wedge (Det(e) \vee e$ minimal promised event); and we prove that $(s_i, c_i) \xrightarrow{e} (s'_i, c'_i)$ and $c'_i = \{e | e \in \mathcal{C}'_p \wedge e \in P_i\}$.

Concerning $c_i^{\sqcap} \subseteq s_i$, in all cases, Property F.2 ensures that $c_i^{\sqcap} \subseteq s_i$, because, as soon as we only apply reductions **(1)**, **(5)**, and **(6)**, all elements of $ci \backslash cuts_i$ are promised events.

- If $Det(e)$ then $\mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}'$ (Definition 12), thus $c_i^{\sqcap} \xrightarrow{e} c''_i$; as $c_i^{\sqcap} \subseteq s_i$ we necessarily have $s_i \xrightarrow{e} s'_i$ (Property F.1). Let $\mathcal{P}(e') \in c_i$ then $\mathcal{P}(e') \in \mathcal{C}_p$, by definition $\mathcal{P}(e') \notin \mathcal{C}_p^{\sqcap}$; but $\mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}'$ thus for any potential causality order $\mathcal{P}(e') \not\prec e$. Finally, $\nexists \mathcal{P}(e') \in c_i$, $\mathcal{P}(e') \prec'_i e$. Which ensures $(s_i, c_i) \xrightarrow{e} (s'_i, c'_i)$.

  $c'_i = c_i \cup \{e\}$ ensures $c_i = \{e | e \in \mathcal{C}_p \wedge e \in P_i\}$, as $\mathcal{C}'_p = \mathcal{C}_p \cup \{e\}$. Concerning the local ordering, the main difference between the two orders is for elements that are in s_i but not in $c_i^{\sqcap}$. Let $e' \in s_i \backslash c_i^{\sqcap}$; $Det(e)$ and e can be executed after $c_i^{\sqcap}$ implies that e is minimal in $s_i \backslash c_i^{\sqcap}$.

- $\neg Det(e)$ then there is $\mathcal{P}(e')$ minimal in $\mathcal{C}_p$, $\mathcal{P}(e') \lhd e$, moreover, we showed in the proof of Property 3.4 that $\mathcal{C}_p^{\sqcap} \xrightarrow{e} \mathcal{C}'$, thus in the same way as in the deterministic case $c_i^{\sqcap} \xrightarrow{e} c''_i$, and thus $s_i \xrightarrow{e} s'_i$ (Property F.1[2]). Thus reduction **(5)** can be applied, and $(s_i, c_i) \xrightarrow{e} (s'_i, c'_i)$.

  Moreover, $c'_i = c_i\{\mathcal{P}(e') \leftarrow e'\} = \{e | e \in \mathcal{C}'_p \wedge e \in P_i\}$, and concerning the local ordering, the fact that in both cases the promised is replaced by the real event is sufficient to conclude.

Moreover, note that the execution occurring after the no dead-lock execution is such that $\mathcal{C}_p$ is a consistent cut, without any promised event, so either $Det(e)$ and reduction **(1)** applies, or $\neg Det(e)$ and reduction **(6)** applies trivially. □

Property F.5 *Every non-triggerable event can be considered as a triggerable one.*

This property are proved by verifying that, if $(s_i, c_i) \xrightarrow{e} (s'_i, c'_i)$ by the reductions **(5)** or **(6)**, and on has $Trig(e)$ then $(s_i, c_i) \xrightarrow{e\mathcal{T}(e)} (s'_i, c'_i)$ by the reductions **(2)** or **(3)** plus the reduction **(4)**. One mainly has to verify that the prerequisites for reduction **(5)** ensures **(2)** and **(4)** can be applied, and the same for **(6)** and **(3)** plus **(4)**.

This property ensures that the three rules **(2)**, **(3)**, and **(4)** supposing $Trig(e)$ are not necessary in Definition 14. However, these rules provide much more asynchrony for the reduction.

The last property necessary to prove the correctness of the local recovery mechanism is the following:

[2]Recall that this necessitates that in-transit messages are replayed at re-execution

Property F.6

$$(s_i, c_i) \xrightarrow{e^1..e^n} (s'_i, c'_i) \Rightarrow \mathcal{C}_p \xrightarrow{*} \mathcal{C}_p^1 \wedge (s'_i, c'_i) \xrightarrow{*} (s_i^1, c_i^1)$$

where $c_i^1 = \{e | e \in \mathcal{C}_p^1 \wedge e \in P_i\}$

We will concentrate on the following proof, the generalization being easy:

Property F.7

$$(s_i, c_i) \xrightarrow{e} (s'_i, c'_i) \Rightarrow \mathcal{C}_p \xrightarrow{*} \mathcal{C}_p^n \wedge (s'_i, c'_i) \xrightarrow{*} (s_i^n, c_i^n)$$

where, for the initial configuration (s_i, c_i) *we have* $c_i = \{e | e \in \mathcal{C}_p \wedge e \in P_i\}$ *and* $c_i^{\sqcap} \subseteq s_i^2$ *where* $s_i \xrightarrow{\mathcal{T}(e^1)..\mathcal{T}(e^n)} s_i^2$. *And similarly for the final configuration* (s_i^n, c_i^n).

Proof : First, because of Property F.5, it is sufficient to verify the theorem in the case where we would not apply reductions **(5)** and **(6)**. Indeed, reductions **(5)** and **(6)** could be rewritten to some reductions **(2)**, **(3)** and **(4)**; a non-triggerable event is just an event for which we must apply reduction **(4)** right after reduction **(2)** or **(3)**, meaning also that we must be check that **(4)** is applicable before doing **(2)** or **(3)**.

Similarly to the proof of Property F.4, we suppose that for each step of the reduction $c_i = \{e | e \in \mathcal{C}_p \wedge e \in P_i\}$ and $(s_i, c_i) \xrightarrow{e} (s'_i, c'_i)$; and we prove that $\mathcal{C}_p \xrightarrow{*} \mathcal{C}_p^1 \wedge (s'_i, c'_i) \xrightarrow{*} (s_i^1, c_i^1)$ and $c_i^1 = \{e | e \in \mathcal{C}_p^1 \wedge e \in P_i\}$.

Concerning $c_i^{\sqcap} \subseteq s_i^2$ where $s_i \xrightarrow{\mathcal{T}(e^1)..\mathcal{T}(e^n)} s_i^2$; Property F.2 ensures that if $e \in c_i^{\sqcap} \backslash cut(s_i)$ then $e \in c_i \backslash cut(s_i)$ and $Trig(e)$. Moreover, $\forall e' \prec e, e' \in c_i^{\sqcap}$, where $\prec$ is the causal order for the first execution, which is identical to the order on $\mathcal{C}_p$ until the last promised event; let e^1 be a minimal event such that $e^1 \in c_i \backslash cut(s_i) \wedge e^1 \prec e$. The reduction **(4)** labeled $\mathcal{T}(e^1)$ applies to s_i: let $\mathcal{C}_p'' = \mathcal{C}_p \backslash \{e^1\}$, and $c_i'' = \{e | e \in \mathcal{C}_p'' \wedge e \in P_i\}$, by construction, and because e^1 is minimal[3], $\mathcal{C}_p''^{\sqcap} \xrightarrow{e^1} \mathcal{C}'$, and $c_i'' \subseteq s_i$; thus $\exists s_i^3$, $s_i \xrightarrow{\mathcal{T}(e^1)} s_i^3$. Applying the same similarly and recursively for all $e^k \prec e$, one finally obtains s_i^n such that $c_i'^{\sqcap} \subseteq s_i^n$ where $s'_i \xrightarrow{\mathcal{T}(e^1)..\mathcal{T}(e^n)} s_i^n$.

It is important to note that: e minimal in $c_i \backslash cut(s_i) \Rightarrow \nexists \mathcal{P}(e') \in c_i$, $\mathcal{P}(e') \prec'_i e$.
The proof of this implication is trivial. Moreover, the consequence of a non-deterministic event can only occur after this event has been artificially triggered (see the remark below concerning non-deterministic message sending). As triggering an event is subject to wait-by-necessity, this ensures the synchronization relatively to triggerable events.

Remark also that $(s_i, c_i) \xrightarrow{e} (s'_i, c'_i)$ implies $e \notin c_i$.

Suppose now $(s_i, c_i) \xrightarrow{e} (s'_i, c'_i)$, consider the least consistent cut (of the first execution) containing all the events of s_i and of $\mathcal{C}_p^{\sqcap}$, let us denote it $\mathcal{C}$. It is easy to verify that $\mathcal{C} \subseteq \mathcal{C}_p^{\sqcup}$ and, by construction, there are $e^1..e^n$ such that $\mathcal{C}_p^{\sqcap} \xrightarrow{e^1..e^n} \mathcal{C}$. We can choose $e^1..e^n$ to be part of the no dead-lock reduction, or more precisely, to always reduce minimal promised

[3] else one would not have $e^1 \in c_i^{\sqcap}$

or deterministic events $s_i \subseteq \mathcal{C}$ and $e \notin \mathcal{C}$ (because $e \notin c_i$) implies $\mathcal{C} \xrightarrow{e} \mathcal{C}'$. Once again, we rely on the no dead-lock reduction, and on the proof of Property F.4. We then have $(s_i, c_i) \xrightarrow{e^1..e^n} (s_i^n, c_i^n)$. As all the events of the reduction are promised or deterministic ones the reduction diagram commutes, for example:

$(s_i, c_i) \xrightarrow{e^1} (s_i^1, c_i^1)$ and $(s_i, c_i) \xrightarrow{e} (s_i', c_i')$: if $Det(e)$ and $Det(e^1)$ and by definition $\xrightarrow{ee^1}$ is equivalent to $\xrightarrow{e^1 e}$. If $\neg Det(e)$ and $Det(e^1)$ then $s_i' = s_i$, and $s_i \xrightarrow{e^1} s_i^1$, which is sufficient to conclude (see below for the arguments relative to the ordering inside c_i). Else $s_i^1 = s_i$ and the presence of $\mathcal{P}(e^1)$ is sufficient to ensure that the diagram commutes, and the relative position of e^1 and e is not sensible to the order of the two events.

We skip the details about the ordering of events, but it is important to note that deterministic events are ordered relatively to $cut(s_i)$ and not to c_i, thus their position w.r.t. non-deterministic ones depends on the moment the non-deterministic ones are triggered ($\xrightarrow{\mathcal{T}(e)}$), not on the moment $\xrightarrow{e}$ is performed. So, the non-deterministic events are automatically placed after the deterministic ones which are necessary for their execution ($\exists s_i'$, $s_i \xrightarrow{e} s_i'$ implies all the deterministic events that are causally before e must have happened).

Note there is no restrictions on the link between intermediate local configurations and global ones. □

The communication mechanism for recovery is slightly different if the sending of a message is a non-deterministic triggerable event, indeed in that case: a message is still automatically received some time after having been sent, but it is only sent once the $\mathcal{T}(e)$ execution has been performed:

$$s_i \xrightarrow{\mathcal{T}(e)} s_i' \wedge (e, e') \in \Gamma \Rightarrow s_j \xrightarrow{e'} s_j', \text{ with } sj \in S'', \ c' \xrightarrow{*} S''$$

There might still be constraints on the receiving state S'' depending on the causal ordering of messages.

This is necessary in order to ensure the last property: as stated in the proof, a triggerable event occurs inside the state and takes it place only upon the application of the reduction **(4)**, not upon reduction **(2)** or **(3)**.

Finally, Theorem 3.3 is a direct consequence Theorems 3.1 and 3.2, and Properties F.4 and F.6.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399